Brain-Inspired Computing (SS 2017) – **Python Exercise Sheet**

Due date: None

0.1 Introduction to Python

a) Install Python 2.7 on your computer (c.f. `https://wiki.python.org/moin/BeginnersGuide/Download`).

b) If you have not yet worked with Python, reproduce the first 7 Chapters of the Python tutorial `https://docs.python.org/2.7/tutorial/`. (You can omit sections 2.2, 4.7, 4.8, 5.8, 6.4 since they discuss more advanced topics)

c) Implement the Seive of Eratosthenes `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`, an ancient algorithm for finding prime numbers.

d) Using your implementation, find the sum of all the primes below two million (`https://projecteuler.net/problem=10`).

Solution:

```
"""Solution to exercise 0.1"""
import math

def sieve(n):
    """Return a list of boolean values of length n indicating whether the integers
    from 1 to n are primes or not"""
    A = [True for _ in range(0, n)]
    A[0] = False
    A[1] = False

    for i in range(2, int(math.sqrt(n))+1):
        if A[i] is True:
            j = i*i
            while j < n:
                A[j] = False
                j += i

    return A

MAX = 2000000
result = 0
for val, isprime in enumerate(sieve(MAX)):
    if isprime:
        result += val

print result
# -> 142913828922
```

0.2 Values and References

The difference between assignment by reference or assignment by value in programming languages can be explained using the following analogy: When browsing the Internet, you might hit a page that you want to store for later use. In this case, you can either bookmark it (i.e., save its URL to your disk) or save the complete page to your disk.

This distinction exists in high-level programming languages as well. There, if you assign a variable to another variable you can either copy the value or copy the memory location address.

a) Describe the advantages and disadvantages of both behaviors in terms of memory usage, data consistency and the posibility that the original data source might disappear.

In Python, variable assignments are by reference [1] by default (except for number types, such as bool, int and float).

As examples, take the following exerpts from a Stackoverflow question `https://stackoverflow.com/questions/13530998/python-variables-are-pointers`:

```
i = 5      # create int(5) instance, bind it to i
j = i      # bind j to the same int as i
j = 3      # create int(3) instance, bind it to j
print i    # i still bound to the int(5), j bound to the int(3)
```

```
i = [1,2,3]    # create the list instance, and bind it to i
j = i          # bind j to the same list as i
i[0] = 5       # change the first item of i
print j        # j is still bound to the same list as i
```

However, by using the `copy` module, you can enforce assignment by value. The distinction between value and reference assignment is reflected in the existance of the two comparison operators `==` and `is`. The `==` operator compares the values of two variables while the `is` operator checks whether two variables reference the same object. Given the following definitions:

```
import copy

a = [1, 2, 3, 4, 5]
b = a
c = copy.deepcopy(a)
```

Write down the results of the following statements (without using your Python interpreter):

b) `print a == b, a == c, b == c`

c) `print a is b, a is c, b is c`

Now, we'll change `a`: `a[0] = 42`. What do the statements now result in:

d) `print a == b, a == c, b == c`

e) `print a is b, a is c, b is c`

f) `print a[0], b[0], c[0]`

g) What's the difference between deepcopy and copy? Could we have used copy for our example as well?

[1] In other programming languages (like C++), there is a distinction between references and pointers. In Python variable hold what is called a reference to an object. They should however not be mistaken for C++ references or pointers. See `http://scottlobdell.me/2013/08/understanding-python-variables-as-pointers/` and `https://www.tutorialspoint.com/cplusplus/cpp_references.htm`.

Solution:

```
import copy

a = [1, 2, 3, 4, 5]
b = a
c = copy.deepcopy(a)
#c = copy.copy(a)

print a == b, a == c, b == c
# -> True True True
print a is b, a is c, b is c
# -> True False False

a[0] = 42
print "Changing a"
# -> Changing a

print a == b, a == c, b == c
# -> True False False
print a is b, a is c, b is c
# -> True False False
print a[0], b[0], c[0]
# -> 42 42 1

# g) c = copy.copy(a) gives the same results
# Counter-example:
d = [a, c]
e = copy.copy(d)
f = copy.deepcopy(d)

print "At first they are all equal"
print "d", d
print "e", e
print "f", f
# -> d [[42, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
# -> e [[42, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
# -> f [[42, 2, 3, 4, 5], [1, 2, 3, 4, 5]]

e[0][0] = 17
f[0][0] = 23

print "Changing e changes d, changing f does not"
print "d", d
print "e", e
print "f", f
# -> d [[17, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
# -> e [[17, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
# -> f [[23, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
```

0.3 Matrix multiplication and numpy

This exercise introduces you to a very important Python package called `numpy`. With it, you can create and efficiently handle large, multi-dimensional arrays and do mathematical operations on them.

Your task will be ot compare the run-time of the matrix multiplication in numpy with an implementation that you will do yourself.

a) Write a function that creates a random NxM matrix as a *numpy* array (see `https://docs.scipy.org/doc/numpy/user/basics.creation.html#arrays-creation` and `https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randint.html#numpy.random.randint`)

b) Implement a matrix multiplication using `for` loops.

c) Compare your implementation's results with that of numpy (`https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html`). Hint: Numpy arrays can be compared with the `==` operator, using their `all()` method.

d) Measure the runtime of both implementations. For this, you can either use the `timeit` package or the following approach (you might have to introduce some repetitions to get a meaningful measurement):

```
import time

start = time.time()
print("hello")
end = time.time()
print(end - start)
```

Solution:

```
import timeit
import numpy as np

def create_linear_filled(N, M):
    """Create an NxM matrix which is filled with linearly increasing values.
    According to Wikipedia, N is the number of rows and M is the number of
    columns.
    """
    ret = np.empty((N,M))
    for row in range(N):
        for col in range(M):
            ret[row][col] = row*M+col

    return ret

def create_random_matrix(N, M, maxint):
    return np.random.randint(1, maxint, size=((N,M)))

def matrixmult(A, B):
    sh_a = A.shape
    sh_b = B.shape
    assert sh_a[1] == sh_b[0]
    C = np.empty((sh_a[0], sh_b[1]))
    for row in range(sh_a[0]):
        for col in range(sh_b[1]):
                C[row][col] = 0
                for step in range(sh_a[1]):
                    C[row][col] += A[row][step]*B[step][col]
        return C

if __name__ == '__main__':
    # check correctness first

    N = 10
    I = 8
    M = 7
    maxint = 100
    A = create_random_matrix(N, I, maxint)
    B = create_random_matrix(I, M, maxint)
    C1 = np.matmul(A, B)
    C2 = matrixmult(A, B)
    assert C1.all() == C2.all()

    reps = 10000
    setup = 'import numpy as np;import solution3;'
    setup += 'A=solution3.create_random_matrix('+str(N)+','+str(I)+','+str(maxint)+');'
    setup += 'B=solution3.create_random_matrix('+str(I)+','+str(M)+','+str(maxint)+');'

    T1 = timeit.timeit('np.matmul(A,B)',
                       setup=setup, number=reps)
    T2 = timeit.timeit('solution3.matrixmult(A,B)',
                       setup=setup, number=reps)
    print T1
    # -> 0.0643570423126
    print T2
    # -> 8.64190196991
    print T2/T1
    # -> 134.280595555
```